Online resources:

Good resources for starting with Go:

Tutorials point: <https://www.tutorialspoint.com/go/go_variables.htm>

Sentdex: <https://www.youtube.com/watch?v=G3PvTWRIhZA&vl=en>

Learn Go in one video: <https://www.youtube.com/watch?v=CF9S4QZuV30> – a little bit faster, assumes you’re familiar with another programming language but if you feel comfortable with Matlab this video should be fine

Assume prior knowledge of C/C++/C#/Java/JS:

A tour of Go: <https://tour.golang.org/list>

Go by example: <https://gobyexample.com/> - Useful reference once you become more familiar with the general grammar of Go

Go Basics:

Installing/Prereqs:

If you don’t already have Go, go here: <https://golang.org/dl/>and install the correct version under Featured Downloads for your operating system. Don’t install the source file- you’ll have to compile it all yourself. If you’re on mac, use the .pkg file, or .msi if you’re on windows. From golang.org/doc/install:

Mac:

[Download the package file](https://golang.org/dl/), open it, and follow the prompts to install the Go tools. The package installs the Go distribution to /usr/local/go.

The package should put the /usr/local/go/bin directory in your PATH environment variable. You may need to restart any open Terminal sessions for the change to take effect.

Windows:

The Go project provides two installation options for Windows users (besides [installing from source](https://golang.org/doc/install/source)): a zip archive that requires you to set some environment variables and an MSI installer that configures your installation automatically. (Use the MSI installer)

#### MSI installer

Open the [MSI file](https://golang.org/dl/) and follow the prompts to install the Go tools. By default, the installer puts the Go distribution in c:\Go.

The installer should put the c:\Go\bin directory in your PATH environment variable. You may need to restart any open command prompts for the change to take effect.

Test to make sure Go installed properly:

Open up a text editor (any one- can be notepad for right now), save it as main.go and paste the following inside:

package main

import "fmt"

func main() {

fmt.Printf("hello, world\n")

}

Open up the command prompt (or terminal for mac) and ‘cd’ your way into where you saved the main.go file and enter ‘go run main.go’ at the prompt. The window should write ‘hello, world’ back.

Additional ways to run go programs:

You can also compile your go programs to a binary file, which is what we will want to do later for distributing our program and also when the programs get more complicated.

In order to do that, firstly, main.go should be in its’ own folder. Make a new folder using ‘mkdir’ and put main.go inside that folder. Type in ‘go build’ inside the folder where main.go is, and when that finishes, there will be a new binary (executable) file in your folder.

You can either double click on it (from finder on mac or file explorer on windows) which will open a terminal window and run it, or you can enter ‘./hello’, replacing ‘hello’ with whatever the name of the folder is.

Basic Go syntax

Go uses pretty standard characters to do mathematical operations. Add two numbers with +, subtract with -, multiply with \*, and divide with / . Additionally, the Modulo function (remainder after division) is performed using % . For example, 9 % 4 will yield 1.

Just like matlab, = is used to assign, and == is used to test if two things are equal. 1==1 will yield true, 2==1 will yield false.

There are several ways to define a variable in Go. Because Go is a strongly-typed language, most of the time you have to declare exactly the *type* that the variable is. Both of the following will do the same thing:

var a int = 1

a:=1

Both of the above will create a new variable called ‘a’ which stores the integer 1. However, there several caveats with the second approach.

Firstly, you can only assign a variable this way *inside of a function*. Outside of a function, you **must** use the ‘var’ syntax.

Secondly, Go will assign that variable a type based on what you give it. Because we gave it ‘1’, Go will automatically assume your variable to be an integer if you don’t specify. ‘int’ is actually the same as ‘uint’, which is an unsigned integer. That means that our variable a will never be able to be negative. Thus, if you assign it using := and later realize you need a to become -1, Go will have to completely redefine a, which is slow.

Additionally, if you later want to represent decimals of any kind, Go will again have to reassign the variable. Further, if you divide an integer, you’ll get the floored division. Thus, 9/4 == 2, not 2.25.

If you want to capture decimals, you can declare a using a:=9.0, and then a/4 ==2.25 but it will assume a is a 64-Bit floating point number, when you could easily use a 32-Bit floating point number which will take up half of the space.

In general, only use := when you know exactly what the type assigned will be.

The numeric types in Go are as follows:

|  |  |  |
| --- | --- | --- |
| Type | Go syntax | range |
| Unsigned 8bit int | uint8 | [0,255] |
| Unsigned 16bit int | uint16 | [0,65535] |
| Unsigned 32bit int | uint32 | [0,4294967295] |
| Unsigned 64bit int | int64 | [0 to ~1.845x1019] |
| Signed 8bit int | int8 | [-128,127] |
| Signed 16bit int | int16 | [-32768, 32767] |
| Signed 32bit int | int32 | [-2.147x109, 2.147x109] |
| Signed 64bit int | int64 | [-9.223x1018,9.223x1018] |
| Floating 32bit number | float32 | complicated |
| Floating 64bit number | float64 | complicated |
| Complex 64bit number | complex64 | Same as float32 with imaginary |
| Complex128bit number | complex128 | Same as float64 with imaginary |

Also note that byte() is the same as uint8 and rune() is the same as int32

Additionally, there’s a few other datatypes in go like strings, booleans (true/false), and Array/Slices (like lists in python or like a row-vector in matlab)

Structuring Go code

Any code that you run has to be in ‘package main’, and must have a function called main().

Looking back at the hello world example:

package main

import "fmt"

func main() {

fmt.Printf("hello, world\n")

}

‘package main’ tells the compiler that this is code that we want to be able to run (libraries that you import will have different packages)

import is optional. Here we import another package that is built-in to Go called ‘fmt’, which stands for ‘format’. Fmt has several functions for printing lines of text out to the command line.

Func is the keyword you use when you want to define a function.

A function is a block of code that will run every time you enter its’ name. If I write a function called add, all of the code inside of ‘add’ will run every time I enter ‘add()’. Thus, use the open-close parentheses when you want to call a function.

Every code that you want to run using ‘go run ...’ MUST have a main() function. In addition, the main function cannot take any inputs and cannot output anything. Everything happens inside main. When main finishes, the program quits.

When using packages, you can ‘call’ a function from a package using ‘packagename.function()’. When we wrote fmt.Printf(“Hello, world\n”), we called the ‘Printf’ function from the ‘fmt’ package.

Fmt.Printf takes a string as an input. The “Hello, world\n” is the string that we are giving it, and that’s why it prints out “Hello, world”. You can try taking out the \n inside the string- \n is an ‘escape character’ that tells Printf to put a new line after it finishes writing “Hello, world”. If you omit it, the command line’s cursor will be on the same line as “Hello world”.

Additionally, notice the curly braces after main () and at the end. The braces tell the compiler where the function starts and ends. Thus, when you enter ‘go run main.go’, it will automatically run main(), performing every line between the curly braces.

Defining other functions

You can define other functions using

func name(input inputtype) returntype {

…..code to run….

}

where ‘name’ is the name of the function, ‘input’ is the name of the input, ‘inputtype’ is the type that the input will be (int, string, float32, ect), and returntype is the type that your function will return.

For example, to write a simple add function, the code would look something like:

func add(param1 int, param2 int) int {

return param1 + param2

}

Again, we define a new function using func, name the function ‘add’, we take two inputs which we will call ‘param1’ and ‘param2’ which are both integers, and we will return an integer.

‘return’ will end the function, and return the value that comes after to whatever called ‘add’. Here, we return param1+param2, which is the sum of the two variables we put in. Then, we can go into our main() function, and change it to this:

func main() {

a:=add(4,7)

fmt.Println(a)

}

which will print out 11. add returns 4+7 into the new variable a, and then we print out the value of a.

For loops

Go has for loops just like matlab. There are a few ways to write them. The most general for loop looks something like:

for i:=0; i<5; i++ {

fmt.Println(i)

}

If you run that for loop, go will print out 0, 1, 2, 3, 4. the i:=0 part defines our counter variable i, i<5 sets the condition for the for loop to end, and i++ sets what happens after we go through the for loop each time. In go, i++ is shorthand for i=i+1 or i+=1. All three are the same. So, we set i equal to zero, print its’ value, hit the end of the braces, increment i, and continue until i becomes 5 and the for loop ends. Additionally, you could do fewer arguments, if you like, with blank spaces for the parts you don’t want. For example,

i:=0

for ; i<5; {

fmt.Println(i)

i++

}

will do the exact same thing as the for loop above. We do a semicolon to note that the first (i:=0) part is blank, and then start the curly braces after i<5. Note that the condition *is* mandatory so that the for loop knows when to end.

Also, if you are omitting both the first and third parts, you don’t need semicolons at all. You can simply use

for i<5 {

Which will do the same thing.

While loops

It’s usually a better idea to use a for loop rather than a while loop (since a while loop will run forever if you make a mistake) but sometimes you need to use a while loop if you don’t know how many times you need the loop to run. In this case, Go doesn’t have a separate ‘while’ command. Instead, you pass a for loop with no argument at all:

for {

will run forever until you close the program some other way, or you use the ‘break’ command, which will break out of the for loop.

Variadic functions

If you want, you can accept multiple input variables. For example, if we wanted the ‘add’ function to be able to add any number of variables, you use an ellipsis when defining it:

func add(params ...int) int {

sum:=0

for i:=0;i<len(params);i++ {

sum+= params[i]

}

return sum

}

Will return the sum of any numbers you pass in. If in main() we call add(3,4,5,6,7,8), it will return the sum of all those numbers.

Note here that we get each value of ‘params’ using a for loop. With an array (which params is, since there are more than one inputs going into it), you can get individual elements with the [] bracket operator.

Also, note that unlike matlab, Go is a 0-index language, meaning the first position in a list is the 0th position, not the 1st position. Thus, params[0] would be the first of the numbers passed- 3, in the example above. This zero indexing is convenient sometimes for math/algorithmic purposes or when dealing with tables (your 0th row would be your column titles, and then the 5th row would be indexed using [5], ect). Sometimes it’s easier to think of the index as the ‘distance away from the first element’, instead of the element number. Thus, the first element is 0 numbers away from itself, the second element is 1 number away from the first, ect.

Additionally, there’s another way we could have written the above function:

func add(params ...int) int {

sum:=0

for i:= range params {

sum+= params[i]

}

return sum

}

Note that we replaced all the for loop conditions with i:= range params. That will do the same thing as before, iterating through all the parameters we give it. This isn’t so interesting, but you can do something else with range:

func add(params ...int) int {

sum:=0

for \_,v:= range params {

sum+= v

}

return sum

}

Now we changed the for condition with \_,v:= range params. Range actually outputs two things on every iteration of the for loop: the index *and* the value. Since we only really need the value here, we use an underscore to say that we want to ignore the index. Then we can simply add v to the sum every time instead of going back and indexing each parameter with [i] separately.

Note that if you leave the I without using the underscore, the compiler will give an error because we don’t ever use I. Go will refuse to compile if you use extraneous variables or imports, and will give you an error until you get rid of them. It’s more efficient to make sure that we aren’t declaring variables that we aren’t using in our program.

Arrays

Just like defining other variables, we define an array using var or the := command. However, when we define an array, the type goes next to the array. For example:

var countingarray = []int{1,2,3,4,5}

countingarray:=[]int{1,2,3,4,5}

will both declare the same variable. Note that we use curly braces to surround the variables, and that we still have to define the type when we use :=. Go won’t assume the type of an array.

Then, if you index the array with countingarray[2] like we did above, Go will print out 3 (Remember, 0 is the index of the first element).

If you don’t know what you want to put in the array yet, you can preallocate an empty array using a:=make([]int, 5) which will make an array of integers of size 5. You can individually index each part to assign values. For example, a[1]=4 would set the second element to be 4.